MASTER EN SCIENCES INFORMATIQUES

Une analyse du macro-assembleur PEGASE

Montigny, Roch

Award date:
1980

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B - 5000 NAMUR (BELGIUM)

UNE ANALYSE DU
MACRO-ASSEMBLEUR PEGASE

*

Roch MONTIGNY

Mémoire présenté pour l'obtention du grade
de Licencié et Maitre en
Informatique

1980

"Avant d'introduire ce travail, je tiens à remercier ici Monsieur le professeur C. CHERTON, promoteur de ce mémoire, pour les conseils et les encouragements qu'il m'a prodigués.

Je remercie Patrick GARDIN pour son aide.

Je remercie enfin tous ceux qui m'ont permis de mener à bien ce travail."

1.TABLE DES MATIERES

2.OBJECTIF DU MEMOIRE

Le but de ce mémoire est de faire une première analyse d'un interpréteur MAGE: interpréteur de programmes écrits en langage MAGE.

En partant du rapport "PEGASE", où nous trouvons les spécifications de MAGE, nous essayerons de mettre en évidence les structures de données utilisées par l'interpréteur, de même que les différents traitements éffectués par celui-ci.

Nous commencerons par montrer pourquoi nous avons décidé de faire de MAGE un interpréteur.
Dans le même chapitre nous décrirons rapidement comment nous envisageons le passage entre BAS et MAGE.
Apres cela nous rappellerons les types de structures de données que nous utiliserons ici: les piles, les listes et les arbres.
Nous envisagerons ensuite le langage intermédiaire IMAGE et nous dégagerons ce qu'il apporte à l'analyse de l'interpréteur. Nous verrons également les contraintes supplémentaires qu'imposent les spécifications du langage MAGE, en même temps que la description des objets nécessaires à l'interprétation.
Enfin, grâce aux concepts dégagés de IMAGE et de MAGE, nous proposerons une approche algorithmique de l'interpréteur.

3.PRECISIONS

3.1 Interpréteur ou traducteur

On peut distinguer deux méthodes pour exécuter un programme écrit dans un langage donné: la traduction et l'interprétation.
La traduction consiste à traduire le programme du langage donné dans un langage machine. C'est ce programme traduit dans le langage machine qui sera exécuté. L'interprétation par contre est une exécution directe du programme. L'interpréteur lit le texte du programme tel qu'il a été écrit et exécute ses instructions. Il n'y a donc pas cette phase de traduction qui fait passer le programme d'un langage source dans un langage cible. Cependant un texte doit être interprété chaque fois qu'il est exécuté, alors qu'un texte ne doit être traduit qu'une seule fois: en effet, la traduction d'un même programme donne toujours le même texte et cette traduction peut être exécutée plusieurs fois.

Certains langages permettent la génération de textes écrits dans le même langage. Le texte généré apparait lors de l'exécution du programme et fait partie alors du programme. Ainsi le texte généré doit lui-même être exécuté.
Si l'exécution du programme se fait par un interpréteur, il suffit de faire interpréter le texte généré au moment où il doit donner lieu à exécution. C'est pourquoi MAGE est un interpréteur: l'évaluation d'un ‹op-call› peut générer du texte en langage MAGE; il suffit de substituer le texte généré à l'‹op-call› qui l'a généré puis de reprendre l'interprétation au début de ce texte.

3.2 Passage BAS-MAGE

Nous abordons ici le problème de l'interface entre l'interpréteur MAGE et l'assembleur BAS. L'interface est assuré d'une part par l'opérateur BAS (PEGASE p.30) et d'autre part par la table des symboles.

Puisque l'interpréteur MAGE est un macro-assembleur, le résultat d'une interprétation d'un programme MAGE est un programme en code machine. Ce code machine résulte de l'assemblage d'un texte en langage BAS.
La fonction d'assemblage est assurée par l'opérateur BAS du langage MAGE; chaque fois que l'opérateur BAS est appelé, son argument est traduit en code machine et joint au code déjà produit ainsi.

L'assembleur BAS utilise des adresses symboliques et doit posséder une table des symboles. On peut envisager une table des symboles unique pour l'assembleur BAS et pour l'interpréteur puisque toute assignation explicite d'une valeur à un symbole utilisé par BAS est faite au niveau du langage MAGE. La seule assignation que BAS fasse est l'assignation implicite de ses labels.
Remarquons d'ailleurs que les labels du langage BAS indiquent des positions dans le code machine produit par l'interprétation tandis que ceux du langage MAGE ont un sens à l'interprétation: ils indiquent une position dans le texte à interpréter.

4.RAPPEL DE QUELQUES STRUCTURES DE DONNEES

Nous rappelons ici les trois types de structures de données que nous utiliserons par la suite. Il s'agit des piles, des listes et des arbres.

-piles

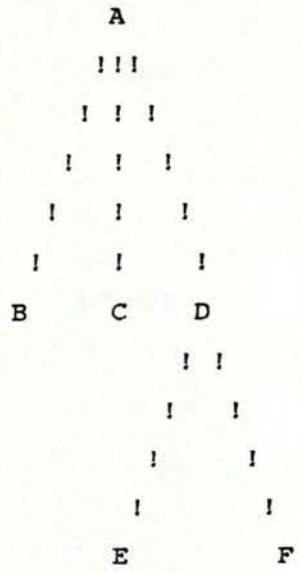
Une pile est une suite d'éléments telle qu'à tout moment nous ne pouvons introduire un nouvel élément qu'à la suite du précédent introduit et nous ne pouvons retirer que le dernier élément introduit.

-listes

Une liste sera également une suite d'éléments mais ici l'insertion ou le retrait d'un élément peut se faire n'importe où dans la liste. Chaque élément d'une liste pourra être désigné par un pointeur.

-arbres

Rappelons enfin qu'un arbre est souvent considéré comme constitué d'une racine et d'un ensemble de sous-arbres. L'ensemble de ces sous-arbres constitue une forêt (qui peut être vide) dont chaque arbre est constitué a son tour d'une racine et d'une forêt. De ce point de vue, une feuille est un arbre dont la forêt est vide.(figure 4.1)



(A,B,C,D,E,F): un arbre

A: racine de l'arbre (A,B,C,D,E,F)

((B),(C),(D,E,F)): forêt de l'arbre (A,B,C,D,E,F)

(B), (C), (E) et (F) : arbres dont la forêt est vide

(D,E,F) arbre dont la forêt est non vide.

figure 4.1

5.APPORTS SPECIFIQUES DU LANGAGE IMAGE A L'ANALYSE DE L'INTERPRETEUR MAGE

5.1 Introduction

Le but de ce chapitre est de mettre en évidence les contraintes spécifiques qu'apporte le langage IMAGE à l'interpréteur.
Nous ne voulons pas décrire ici un interpréteur particulier pour IMAGE mais seulement montrer quels aspects de l'interpréteur, dont nous faisons l'analyse, sont dûs aux spécifications de IMAGE.
Un interpréteur propre au langage IMAGE serait sans doute différent de ce qui sera dit ici. Ce chapitre n'est qu'une étape intermédiaire dans la construction de l'interpréteur MAGE.
Lorsque plus tard, nous envisagerons l'apport du langage MAGE, nous repartirons des concepts dégagés ici.

Nous diviserons le chapitre en deux parties. Dans la première partie, nous utiliserons la notion d'arbre pour comprendre la structuration d'un programme et le cheminement de l'interpréteur à travers ce programme.
Dans la seconde partie, nous verrons quelles fonctions l'interpréteur doit réaliser et quelle structure de données il suffit d'implémenter pour répondre aux spécifications de IMAGE.

5.2 Première partie

Nous commencerons par la description statique d'un programme, puis nous verrons comment dynamiquement ce programme est interprété.

5.2.1 Aspects statiques

Rappelons d'abord que ‹image-programme› désigne aussi bien le programme dans son ensemble que le texte associé à un opérateur de type $iprg ou $iprgv.

5.2.1.1 Un ‹op-call› est un arbre

D'après la syntaxe de IMAGE, un ‹arg› un ‹oprt-id› ou un ‹stmnt› peuvent être des ‹op-call›.
Un ‹op-call› est lui-même défini comme étant (PEGASE p.9)

‹(arg-seq›)‹oprt-id›(‹arg-seq›)›
où
‹arg-seq›::=‹arg›!‹arg›‹arg-sep›‹arg-seq›
et où
‹arg-sep›::=,!‹ident›

Nous nous représenterons les ‹op-call› comme étant des arbres où les sous-arbres dont la forêt est vide sont des ‹arg› et des ‹oprt-id› qui ne sont pas des ‹op-call› et où les sous-arbres dont la forêt est non vide sont des ‹arg› et des ‹oprt-id› qui sont des ‹op-call›.
D'une manière générale, les descendants de la racine d'un arbre sont les éléments d'un ‹op-call›.(figure 5.1)
Les caractères alphabetiques majuscules sont des ‹oprt-id› qui ne sont pas des ‹op-call›. (dans l'exemple A et B)
les caractères alphabetiques minuscules sont des ‹arg› qui ne sont pas des ‹op-call›. (dans l'exemple a,b,c et d)
Nous omettrons les points devant délimiter les ‹oprt-id›. Ceci sera justifié au chapitre 6.
Ces conventions seront respectées dans les exemples ultérieurs.

figure 5.1

5.2.1.2 Un <image-programme> est une suite d'arbres

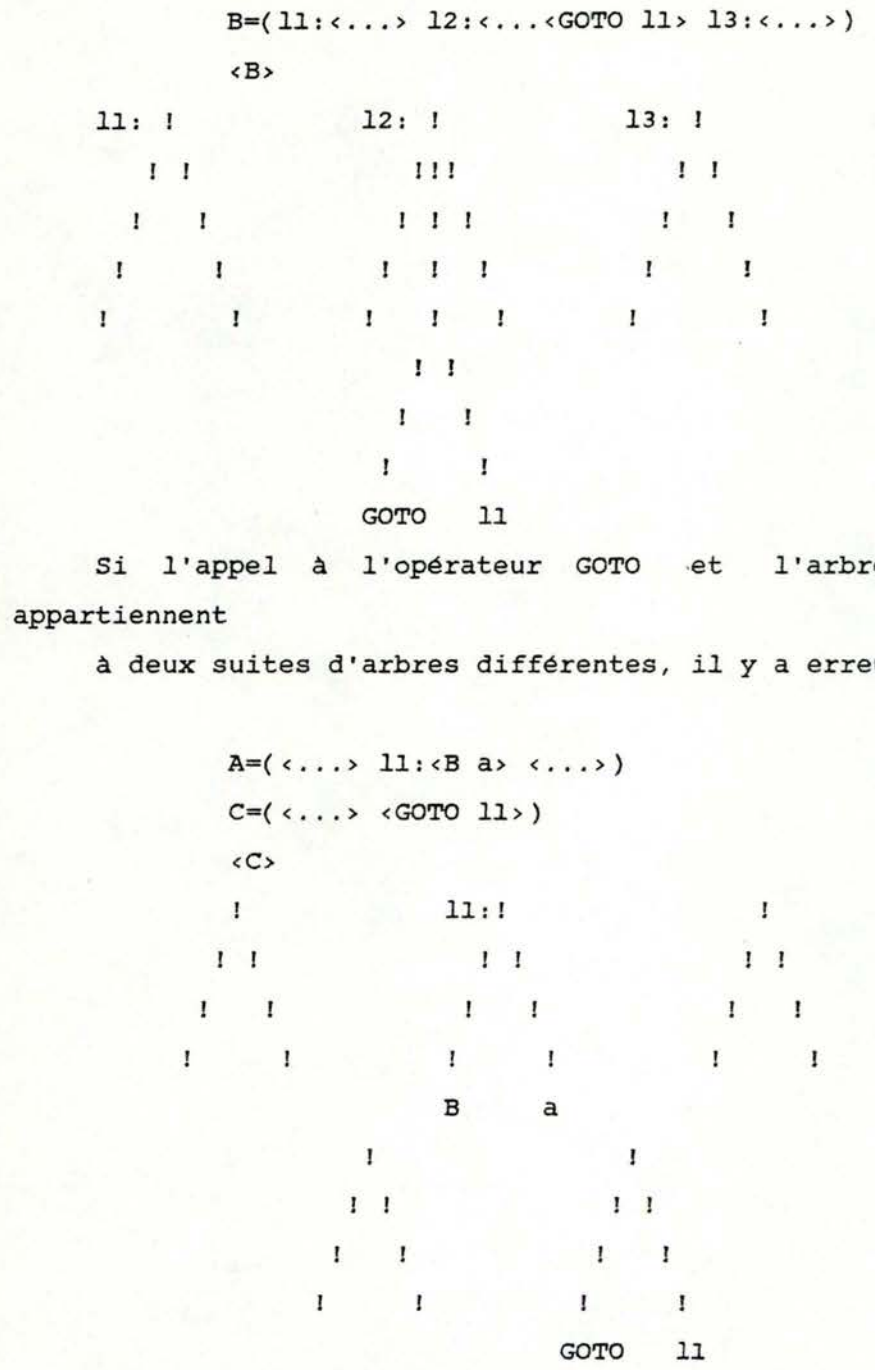
La syntaxe d'un <image-programme> nous rappelle que cet <image-programme> est constitué d'une suite d'<op-call> étiquetés chacun par un <label>. (PEGASE p.9)

<image-programme>::=<stmnt>!<stmnt><image-programme>
<stmnt>::=<label><op-call>
<label>::=<ident>:!<empty>

Nous venons de voir qu'un <op-call> peut être représenté par un arbre: un <image-programme> peut donc être perçu comme une suite d'arbres étiquetés ou encore comme une forêt ordonnée.
Notons ici que l'opérateur GOTO (PEGASE p.26) ne peut référencer que le <label> d'un arbre de la suite auquel l'arbre qui contient cet opérateur GOTO appartient.(figure 5.2)

5.2.2 Aspects dynamiques

5.2.2.1 <image-programme>



Si l'appel à l'opérateur GOTO et l'arbre référencé appartiennent à deux suites d'arbres différentes, il y a erreur.

L'évaluation de <GOTO l1> provoque une erreur.

figure 5.2

Interpréter un <image-programme> revient à évaluer la suite des arbres qui le représente à partir de l'arbre le plus a gauche. Les arbres de la suite sont donc interprétés séquentiellement de gauche à droite. L'opérateur GOTO peut modifier cette séquence.

5.2.2.2 ‹op-call›

L'évaluation d'un arbre est un processus récursif qui consiste à évaluer un ‹op-call› en évaluant d'abord tous les ‹op-call› qui lui sont internes (ces ‹op-call› internes étant des arbres d'une forêt au sens ou nous considérons qu'un arbre possède une racine et une forêt). L'opérateur correspondant à l'‹op-call›, représenté par cet arbre sera donc appelé lorsque tous les ‹arg› ou ‹oprt-id› de cet ‹op-call› qui étaient eux-même des ‹op-call› ont été évalués, c'est-à-dire lorsque la forêt ne contient plus que des feuilles.
En outre si plusieurs ‹op-call› sont simultanément en cours d'interprétation, ils sont toujours imbriqués.
Lorsque l'interpréteur évalue un ‹op-call› arbre de la forêt d'un ‹image-programme›, il cherche à évaluer l'‹op-call› le plus interne le plus à gauche de l'arbre en appelant l'opérateur de l'‹op-call› le plus interne puis il recommence le même processus avec le même arbre jusqu'à ce qu'il l'ait totalement parcouru.

5.2.2.3 Textes générés

L'évaluation d'un ‹op-call› produit un texte, éventuellement vide, qui doit se substituer à l'‹op-call› qui le génère. Ce texte généré est à son tour interprété de suite après la substitution.

Si l'opérateur est un opérateur de base, le texte est généré immédiatement après l'évaluation.
Si l'interpréteur appelle un opérateur IMAGE, évaluer l'‹op-call› revient à interpréter ce nouvel ‹image-programme› constitué lui-même d'une suite d'arbres. Ici le texte se substituant à l'‹op-call› est généré grâce à l'emploi de l'opérateur à:
l'opérateur à permet de générer un texte qui se substitue à

l'‹op-call› d'un opérateur IMAGE. L'appel de à se fait dans l'‹image-programme› de l'opérateur IMAGE.
Le texte généré est l'argument de à.
Un opérateur IMAGE peut utiliser plusieurs fois à. C'est alors la concaténation de tous les textes successivement générés qui forme le texte généré IMAGE, lequel sera interprété lorsque l'interprétation du ‹image-programme› auquel il se substitue sera elle-même terminée.

Exemple 5.3:

...‹A=(...‹à‹(B)›...‹à(C)...)›...

...‹A›...

Apres l'évaluation de ‹A›, ‹A› sera remplacé par BC.

Ainsi se dégagent deux types de textes générés que nous appellerons par la suite texte générés de base et textes générés IMAGE.
Les textes générés de base sont ceux qui se substituent à l'‹op-call› de l'opérateur de base qui les crée.
Les textes générés IMAGE sont ceux qui se substituent à l'‹op-call› de l'opérateur IMAGE qui contient l'appel à l'opérateur de base à grâce auquel ces textes sont générés.

5.3 Deuxième partie

5.3.1 Extraction des atomes

Avant d'évaluer un ‹op-call›, il est nécessaire d'extraire les atomes qui le constituent à partir d'un ‹image-programme›.
Pour certains de ces atomes l'extraction se fait selon la définition d'une classe syntaxique du langage (PEGASE p.12) et on dit que l'atome appartient à cette classe syntaxique:
Les atomes regroupés selon la classe des ‹ident›, ‹string›,

‹num-string› ou ‹oprt-denot› auront la classe syntaxique $ident, $strden, $numden ou $prgden correspondante.
Pour d'autres atomes, l'extraction se fait suivant leur valeur:
-Les éléments de l'ensemble suivant sont des atomes: ($int, $intv, $str, $strv, $lab, $labv, $eprg, $eprgv, $iprg, $iprgv, $lcnt, $lcntv).
Leur classe syntaxique est $spident.
-les ‹op-char›, les ‹, ›, (,), les ‹arg-sep› de la syntaxe les ‹sys-ident› seront chacun des atomes.
Enfin la partie du texte dont les atomes n'ont pas encore été extraits sera considérée comme un atome.

5.3.2 Evaluation des ‹op-call›

5.3.2.1 Chronologie de l'évaluation

Nous savons que dans un ‹image-programme›, l'‹op-call› a évaluer est toujours l'‹op-call› le plus interne le plus à gauche. C'est pourquoi dans le langage IMAGE, l'interpréteur doit évaluer un ‹op-call› chaque fois qu'il rencontre un crochet fermant:
un ‹op-call› est le plus interne le plus à gauche s'il est compris entre le dernier crochet ouvert et le crochet fermant que l'interpréteur vient de lire.

exemple 5.4:

‹(a)A(‹(b)B(c)›,d)›
1 2 3

a,b,c,d sont des arguments qui ne sont pas des ‹op-call›
A,B sont des ‹oprt-id› qui sont des ‹ident› ou des ‹op-char›

Quand l'interpréteur lit le › en position 3, l'‹op-call› à évaluer est ‹(b)B(c)›.

5.3.2.2 Mémorisation des atomes

Il faut mémoriser les atomes, nécessaires à l'évaluation de l'‹op-call› auquel ils appartiennent, entre le moment de leur extraction et le moment où l'évaluation a lieu. Cette memorisation se fait dans une liste dont chaque élément peut être désigné par un pointeur.(5.3.2.3.3)

Y sont mémorisés les atomes appartenant aux classes syntaxiques suivantes: $spident, $ident, $strden, $numden et $prgden. Un élément de la liste contiendra un atome et sa classe syntaxique éventuelle. La classe syntaxique permet de connaitre les types d'objets ($int, $intv, $str, $strv, $lab, $labv, $eprg, $eprgv, $iprg, $iprgv, $lcnt, $lcntv) qui peuvent correspondre à l'atome et ainsi de détecter les erreurs.
Les ‹op-char›, les (, les), et les ‹arg-sep› qui sont des ‹ident› sont également mémorisés dans un élément de la liste.
Il est inutile de mémoriser des crochets rectangulaires ( ) et des doubles quotes (") puisqu'ils délimitent un atome d'une classe syntaxique particulière et que cette classe syntaxique est mémorisée en même temps que l'atome.

Nous mémoriserons dans une pile le pointeur dans la liste du premier élément ajouté à celle-ci après lecture de tout crochet ouvrant. Ceci est nécessaire afin de pouvoir accéder au premier atome du dernier ‹op-call› ouvert: atome contenu dans l'élément pointé par le sommet de la pile lors de la lecture d'un crochet fermant. La structure de pile suffit car les structures de parenthèses ne peuvent être disjointes. Cette pile sera appelée pile des crochets.

En résumé, à l'interprétation d'un ‹image-programme› correspond une structure de données à implémenter que nous désignerons habituellement par le terme de "structure".
Il s'agit

-d'une liste d'éléments

-de la pile des crochets
-du texte déjà généré par l'opérateur à au
cours de l'interprétation de cet ‹image-programme›.

5.3.2.3 Remarques

1.La liste est initialisée avec un seul élément ayant pour atome le programme à interpréter et $iprg pour classe syntaxique.

2.Les parenthèses sont mémorisées dans la liste. Il sera également nécessaire de mémoriser dans la pile des crochets chaque parenthèse ouvrante afin de s'assurer :
-qu'une parenthèse n'a pas été ouverte et non fermée ou qu'un crochet ouvrant manque (...(...>) (parenthèse au sommet de la pile alors que l'interpréteur vient d'extraire un >)
-qu' un crochet n'a pas été ouvert et non fermé ou qu'une parenthèse ouvrante manque (...‹...))(pointeur d'un crochet ouvrant au sommet de la pile alors que l'interpréteur vient d'extraire une parenthèse).
Ceci suppose que les parenthèses pourront être distinguées des pointeurs des crochets dans la pile des crochets.

Chaque fois que l'interpréteur rencontrera une parenthèse ouvrante il enlèvera la parenthèse du sommet de la pile des crochets. Si cette parenthèse n'existe pas, il y a erreur.

3.Nous avons dit plus haut que des atomes extraits étaient mémorisés dans une liste.(5.3.2.2) S'il ne s'agissait que de vouloir construire un interpréteur pour le langage IMAGE une pile suffirait sans doute. Mais nous voulons construire un interpréteur pour le langage MAGE. Si la notion de liste est trop puissante pour le langage IMAGE, elle se justifiera pour le langage MAGE.

5.3.2.4 Appel des opérateurs

Lorsqu'un opérateur est appelé il est nécessaire de connaitre l'adresse dans la liste de l'‹oprt-id› et des atomes extrêmes droits et gauches de l'‹op-call›.
Le pointeur de l'atome extrême gauche est au sommet de la pile des crochets. L'atome extrême droit est l'avant dernier atome de la liste (le dernier étant le texte non encore interprété).
L'‹oprt-id› est connu en balayant la liste entre les deux atomes extrêmes de l'‹op-call› à la recherche d'un atome ‹op-char› ou ‹ident› hors de toute structure de parenthèses et dont le type est $iprg, $iprgv, $eprg ou $eprgv. Ce type est connu en consultant la table des symboles.

L'‹oprt-id› qui serait compris dans une structure de parenthèses serait l'‹oprt-id› d'un ‹op-denot› et non d'un ‹op-call›.

exemple 5.5:

‹A((a)B(c),d)›

(a)B(c) est un ‹op-denot›.

Remarquons qu'il faudrait balayer toute la liste entre les deux atomes extrêmes afin de s'assurer qu'il n'existe qu'un ‹oprt-id› possible pour l'‹op-call›, sans quoi il y a erreur. Ceci ne sera plus vrai pour MAGE.(6.4.2.3)

5.3.3 Modification de la liste après évaluation d'un opérateur de base

Certains opérateurs de base (+, -, *, /, ', SEQL, STRING, SUBSTRG, STRLG, NUMSTR, COMP) génèrent des ‹sys-ident›.
L'opérateur ARG (PEGASE p.20) génère des textes qui ne peuvent être des ‹image-programme› et l'opérateur VAL (PEGASE p.22) des

textes qui peuvent être des ‹image-programme›.

5.3.3.1 Génération d'un ‹sys-ident›

Si un opérateur génère un ‹sys-ident›, il suffit à l'opérateur de substituer dans la liste l'‹op-call› par l'atome ‹sys-ident›.

exemple 5.6:
Supposons que ‹(b)B(c)› génère un ‹sys-ident› que nous appelerons sysid.
Apres l'évaluation de (b)B(c)› l'‹image-programme› ‹(a)A(‹(b)B(c)›,d)› devient ‹(a)A(sysid,d)›.

5.3.3.2 Opérateur ARG (PEGASE p.20)

L'opérateur ARG génère dans l'‹image-programme› en cours d'interprétation un des arguments de l'‹op-call› dont l'‹oprt-id› désigne cet ‹image-programme›. Il suffit dans ce cas de substituer à l'‹op-call› dont l'‹oprt-id› est ARG la suite des éléments qui constitue cet argument.

Exemple 5.7:

‹A=(‹‹ARG (1)››...)›

‹A((a)B(c),d)›

Soit A l'‹image-programme› en cours suite à l'‹op-call› ‹A((a)B(c),d)› Apres l'évaluation de ‹ARG(1)›, l'‹image-programme› devient ‹(a)B(c)›.... .

5.3.3.3 Opérateur VAL (PEGASE p.22)

L'opérateur VAL génère un ‹string›. Ce ‹string› est soit le ‹string› du ‹string-denot› argument de VAL soit la valeur assignée à l'‹ident› argument de VAL. Dans ce cas, l'‹ident› est un objet de type $str ou $strv et la valeur de l'‹ident› est mémorisée dans la table des symboles. Dans les deux cas, les différents atomes du ‹string› n'ont pas encore été extraits. (En effet, jusqu'ici le ‹string› a été considéré globalement comme un atome de la classe syntaxique $strden.) C'est pourquoi le ‹string› préfixe la partie du texte non encore interprété de l'‹image-programme›, tandis que l'‹op-call› de VAL disparait de la liste. L'interprétation reprendra au début du ‹string›.

Exemple 5.8:

```
‹A="string">
‹VAL (A)>‹...>
```

Apres l'évaluation de ‹VAL(A)›, l'‹image-programme› ‹VAL(A)›‹...› devient string‹...›

5.3.4 Modification de la liste suite a l'évaluation d'un opérateur image

Lorsqu'un opérateur IMAGE est appelé, l'interprétation de l'‹image-programme› en cours doit être suspendue. Elle ne sera reprise que lorsque l'interprétation de l'‹image-programme› de l'opérateur IMAGE appelé sera elle-même terminée.

Une nouvelle structure est crée et initialisée pour chaque ‹image-programme› lors de l'appel d'un opérateur IMAGE.
L'imbrication d'appel des ‹image-programme› implique une pile des structures en suspens.

Lorsqu'un ‹image-programme› est entièrement interprété, il faut reprendre l'interprétation de l'‹image-programme› précédent. Ceci est possible en réactivant la derniere structure suspendue.

Cependant, avant de reprendre l'extraction des atomes, il faut substituer la suite des éléments générés par l'opérateur à, au cours de l'interprétation de l'‹image-programme› venant d'interpréter l'‹op-call› de cet ‹image-programme›.

Exemple 5.9:
Soit B l'‹oprt-id› d'un ‹image-programme› Apres l'évaluation de ‹B› l'‹image-programme› ‹B›‹...› devient:
suite d'atomes générés par à‹...›

Remarque
Nous revenons un instant sur l'opérateur à afin de montrer ce qui le distingue de l'opérateur VAL.
D'abord le texte généré par l'opérateur VAL remplace l'‹op-call› de cet opérateur ce qui n'est pas le cas pour l'opérateur à.
Ensuite, suivant le rapport PEGASE, VAL génère le ‹string› désigné par son opérande droit (PEGASE p.22) alors que l'opérateur à génère une séquence quelconque d'arguments droits. (PEGASE p.22) Ceci signifie que si à désigne par son operande un ‹string›, c'est ce qui désigne ce ‹string› qui sera généré et non pas la valeur du ‹string›.
Exemple: ‹string ="texte"›
‹à(string)› génère string
‹VAL(string)› est remplacé par texte.

5.3.5 Dynamique des structures

5.3.5.1 Définitions

Nous considérons les listes comme des suites d'éléments chainés les uns aux autres.
Un élément pointe vers l'élément qui le précède et vers celui qui le suit dans la liste.

Exemple 5.10:

__élément 1__élément 2__élément 3__

représente trois éléments chainés dans une liste

L'élément 1 a pour pointeur a.

L'élément 2 a pour pointeur b.

L'élément 3 a pour pointeur c.

Le pointeur de l'élément précédent l'élément 2 sera a:=précédent(b).

Le pointeur de l'élément suivant l'élément 2 sera c:=suivant(b).

L'atome mémorisé par l'élément dont le pointeur est a est atome(a).
La classe syntaxique, si elle existe, mémorisée par l'élément dont le pointeur est a, est classe syntaxique(a).

Par position courante nous désignerons le pointeur du dernier élément d'une liste.
Puisqu'il existe une liste par structure, chaque structure aura sa position courante.
Le dernier élément de chaque liste est la partie de l'‹image-programme› qui n'a pas encore été lue par l'interpréteur.
Chaque fois qu'un nouvel atome devra être extrait, la position courante permettra d'accéder au texte d'où l'atome doit être extrait.
Si l'atome extrait doit être mémorisé dans la liste, son élément sera inséré avant l'élément pointé par la position courante.

5.3.5.2 Génération d'un ‹sys-ident›(exemple 5.6)

Dans les exemples qui suivent seuls les atomes des éléments ont été représentés; les chiffres représentent des pointeurs.

```
Structure avant l'évaluation de <(b)B(c)>:
liste:   (__a__)__A__(__(__b__)__B__(__c__)__,d)>
         1              2                 4  3
position courante:  3
pile des crochets:  2
                    1
Structure après l'évaluation de <(b)B(c)>:
liste:  (__a__)__A__(__sysid__,d)>
        1                    3
position courante:  3
pile des crochets:  1
```

La position courante reste la même. L'élément sysid se substitue à l'<op-call> compris entre les adresses 2 et 4. Cet atome pointe à gauche vers l'élément dont le pointeur est précédent(2) et à droite vers l'élément dont le pointeur est la position courante.

5.3.5.3 Opérateur ARG (exemple 5.7)

```
Structure avant l'évaluation de <ARG(1)>:
liste:  ARG__(__1__)__>
        1         2
position courante:  2
pile des crochets:  1
                    1
Structure après l'évaluation de <ARG (1)>:
        (__a__)__B__(__c__)__>
        1                   2
position courante:  2
pile des crochets:  1
```

La position courante reste inchangée. L'<op-call> de ARG disparait. La suite des éléments (__a__)__B__(__c__) pointe à gauche vers l'élément dont le pointeur est précédent(1) (ici cet élément n'existe pas) et à droite vers l'élément dont le pointeur

est la position courante.

5.3.5.4 opérateur VAL (exemple 5.8)

Structure avant l'évaluation de VAL:

liste: VAL__(__A__)__...

1 2

position courante: 2

pile des crochets: 1

Structure après évaluation de ‹VAL(A)›:

liste: string__...

3 2

position courante: 3

pile des crochets:

L'extraction des atomes doit reprendre au ‹string› généré par VAL. La position courante est modifiée et pointe vers ‹string›. Nous supposerons provisoirement que lorsque l'élément désigné par la position courante devient vide, cet élément est éliminé de la liste et nous supposerons que le pointeur de l'élément suivant devient le pointeur désigné par la position courante. Ce point sera eclairci lorsque nous examinerons le langage MAGE. (6.4.2.5.2)

5.3.5.5 Insertion du texte généré par l'opérateur à(exemple 8)

Structure lorsqu'elle est suspendue:

liste: ...__B__...

1 2

position courante: 2

pile des crochets: 1

.

.

Structure lorsqu'elle est réactivée:

liste: ...--suite des atomes générés par l'opérateur à--...

2

position courante: 2

pile des crochets: .

.

.

De ces exemples nous dégageons deux points:

-VAL est le seul opérateur pouvant modifier la position courante.

-Chaque fois qu'un ‹op-call› est évalué, le pointeur du sommet de la pile des crochets est retiré.

6. APPORT DU LANGAGE MAGE A L'ANALYSE DE L'INTERPRETEUR MAGE

6.1 Introduction

La notion de nombre maximum d'arguments gauches (droits), la notion de priorité ainsi que cinq règles ajoutées au langage IMAGE définissent le langage MAGE. (PEGASE p.32)
Dans ce chapitre nous allons compléter l'interpréteur MAGE que nous avions décrit au chapitre précédent.
Tenant compte du nombre maximum des arguments, de la priorité des opérateurs ainsi que des cinq règles, l'interpréteur devra être capable d'interpréter un programme écrit dans le langage MAGE.

Dans la première partie, nous préciserons certains points du rapport PEGASE et nous apporterons éventuellement quelques modifications.
Dans la deuxieme partie du chapitre nous expliquerons le passage du langage IMAGE au langage MAGE en termes d'arbres.
Dans la troisieme partie nous verrons l'apport du langage MAGE à l' analyse de l'interpréteur.

6.2 Première partie

6.2.1 Passage du langage IMAGE au langage MAGE

Le but du langage MAGE est d'alléger le texte du maximum de ses crochets et parenthèses. Les notions de nombre maximum d'arguments gauches (droits), de priorité des opérateurs ainsi

que cinq règles ont été introduites afin de permettre à l'interpréteur de retrouver la liste des arguments de chaque ‹op-call› et l'ordre de succession de leur évaluation tel que le même programme écrit en langage IMAGE les définirait. La priorité permet de choisir un ‹oprt-id› particulier parmi un ensemble d'‹oprt-id› possibles. Le nombre maximum d'arguments droits (gauches) permet de retrouver la liste des arguments droits (gauches) d'un ‹op-call› si cette liste n'est pas délimitée par des parenthèses.

6.2.2 Règle 1

La règle 1 ainsi que la règle 2 indiquent dans quelles conditions certains crochets et parenthèses peuvent être omis.

-règle 1 et labels.

La règle 1 ne tient pas compte de la syntaxe des ‹stmnt›.

‹stmnt›::=‹label›‹op-call›

En effet, si nous appliquons la règle 1 à un ‹stmnt› précédé d'un ‹cr› et dont le ‹label› n'est pas un ‹empty›, l'interpréteur considèrera systématiquement que le ‹label› est précédé d'un crochet ouvrant implicite, ce qui est erroné.

Nous proposons donc de modifier la règle 1 comme suit:
Tout ‹crlf› non interne à une paire de crochets se correspondant et non suivi immédiatement d'un crochet ouvrant explicite ou d'une liste de labels et d'un crochet ouvrant explicite est considéré comme suivi immédiatement d'un crochet ouvrant implicite.

-règle 1 et parenthèses

La première règle semble ouvrir un crochet implicite si un ‹crlf› se situe après une parenthèse ouvrante non encore fermée quand cette parenthèse n'est pas comprise entre crochets explictes. C'est pourquoi nous proposons une seconde modification à la règle 1 qui devient:

Tout ‹crlf› interne à aucune paire de crochets ou de parenthèses et non suivi immédiatement d'un crochet ouvrant explicite ou d'une liste de labels et d'un crochet ouvrant explicite est considéré comme suivi immédiatement d'un crochet ouvrant implicite.

6.2.3 Modification des classes syntaxiques et règle 3

La règle 3 nous permet d'omettre les points delimitant les ‹op-char›. En effet, l'ensemble des ‹op-char› est connu par l'interpréteur qui peut ainsi les reconnaitre lorsqu'il les extrait.

Il nous semble que nous pouvons étendre cette règle 3 aux ‹ident› et par conséquent aux ‹op-call› en omettant les points qui les délimitent.

Les symboles de la classe syntaxique $ident désignent des objets des différents types repris par la semantique de IMAGE. ($int, $intv, $str, $strv, $lab, $labv, $eprg, $eprgv, $iprg, $iprgv, $lcnt, $lcntv)

Une déclaration associe à chacun de ces symboles le type de l'objet qu'ils désignent en le memorisant dans la table des symboles. (Si cette déclaration est explicite, le type d'un objet est déclaré par l'opérateur TYPE (PEGASE p.17)) Il suffit dès lors à l'interpréteur de consulter la table des symboles. Un ‹ident› désignera un ‹oprt-id› si le type de l'objet qui est associé est $eprg, $eprgv, $iprg, $iprgv.

Nous proposons une nouvelle définition syntaxique d'un ‹oprt-id›:

‹oprt-id›::=‹ident›!‹op-char›!‹op-call›

La classe syntaxique $oprt propre aux ‹oprt-id› n'existe plus.

Remarquons d'ailleurs que pour les mêmes raisons, les ‹ident› servant d'‹arg-sep› auraient pu être déclarés comme appartenant à un type particulier d'objets, de telle sorte qu'il soit inutile de les délimiter par des /.

6.2.4 Crochets implicites-crochets omis

Au cours du travail nous distinguerons trois types de crochets dans un texte MAGE:
les crochets explicites, implicites et omis. (6.4.1, 6.4.2.1)
Les crochets explicites seront ceux effectivement présents dans le texte.
Les crochets implicites seront les crochets implicitement présents si on applique les règles 1 et 2.
Les crochets omis seront les crochets qui devraient délimiter les ‹op-call› reconstitués par l'interpréteur en appliquant les règles 4 et 5.

Nous verrons par la suite que l'interpréteur sait où sont placés les crochets implicites au moment même où il lit le texte, alors que les crochets omis ne seront découverts que plus tard ce qui justifie la distinction entre crochets implicites et omis.

6.2.5 Correspondance entre programmes MAGE et programmes IMAGE

Au vu des cinq règles énoncées pour définir la syntaxe de MAGE, il semble évident que l'interpréteur du langage MAGE doit pouvoir

interpréter également un texte écrit dans le langage IMAGE. En effet, l'omission de parenthèses et de crochets est toujours facultative et si l'on en omet aucun, le texte vérifie bien la syntaxe du langage IMAGE.
Bien que la syntaxe de MAGE soit issue de la syntaxe de IMAGE nous ne pouvons cependant supposer que tout texte MAGE découle d'un texte IMAGE équivalent. Un exemple suffit pour le demontrer. (Nous dirons qu'un texte IMAGE et un texte MAGE sont équivalents si non seulement le texte MAGE vient d'un texte IMAGE dont on aurait omis des parenthèses et des crochets mais si en plus interprétés tous les deux par l'interpréteur MAGE, ils donnent le même résultat.)

Soit le texte MAGE:
‹cr› A,B,c C d ‹cr› (1)
avec les priorités suivantes pour les ‹oprt-id›

A: 2

B: 1

C: 3

(1) n'a pas de texte IMAGE correspondant;
dans le texte IMAGE ‹(‹A›,‹B›,c)C(d)› (2)
‹B› est évalué après ‹A› et non avant comme dans (1)

Il apparait bien que le texte (2) n'a pas d'équivalent en langage IMAGE. Pourtant ce texte est correct et peut avoir un sens pour l'interpréteur.

6.3 Deuxième partie

Nous étudierons d'abord le passage du langage IMAGE au langage MAGE d'un point de vue statique. Ensuite nous étudierons la dynamique de l'interprétation d'un ‹image-programme› écrit dans le langage MAGE.

6.3.1 Aspects statiques

6.3.1.1 Signification des crochets implicites en termes d'arbres

Nous savons qu'un ‹image-programme› est une suite d'arbres. (5.2.1) Chaque ‹op-call› arbre de cette forêt est délimité par une paire de crochets. D'après les règles 1 et 2 seuls ces crochets peuvent être des crochets implicites.
Aucun sous-arbre ne peut être délimité par des crochets implicites. Autrement dit, un crochet implicite ne peut être contenu entre deux crochets implicites ou explicites.

‹cr›...‹ ›...‹cr› est valide

‹cr›...‹cr›...‹cr›...‹cr› représente trois ‹op-call› disjoints et non un ‹op-call› inclus dans un seul ‹op-call›

‹...‹cr›...‹cr›...› les ‹cr› ne seront pas considérés comme des crochets implicites puisqu'ils sont internes à une paire de crochets explicites.(règle 1)

6.3.1.2 Signification des crochets omis en termes d'arbres.

Si plusieurs ‹oprt-id› à l'intérieur d'une expression entre crochets explicites ou implicites ne sont pas eux-mêmes compris entre crochets ou entre parenthèses, alors les ‹op-call› dont ils sont les ‹oprt-id› sont délimités par des crochets omis.
(Si un ‹oprt-id› est compris entre des parenthèses, il est l'‹oprt-id› d'un ‹op-denot› et non d'un ‹op-call›.

Exemple:

‹...((...)A(...))B(...)...›

A est l'‹oprt-id› de l'‹op-denot› (...)A(...)

B est l'‹oprt-id› de l'‹op-call› ‹(...)B(...)›)

Exemple:

‹(...)A(...)(...)B(...)(...)C(...)›

(...)A(...),

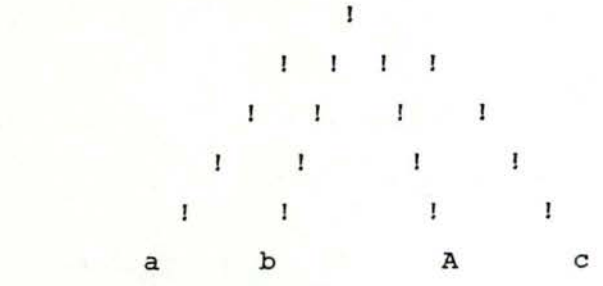
(...)B(...) et

(...)C(...) sont chacun entourés de crochets omis.

Seul un sous-arbre peut être délimité par des crochets omis. Il faut de plus que la forêt auquel ce sous-arbre appartient contienne plus d'un sous-arbre.(figure 6.1)

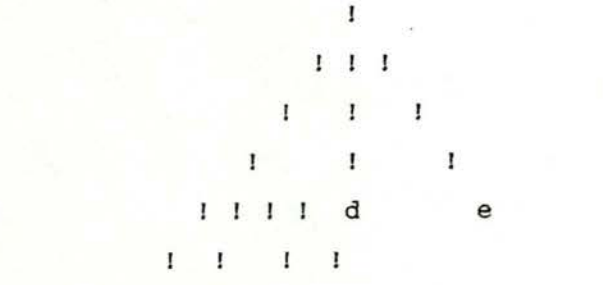
‹cr›(a,b)A(c)‹cr›

(a,b)A(c) n'est pas entouré de crochets omis. L'arbre n'a pas de sous-arbre dont la forêt soit non vide c'est à dire qui soit un ‹op-call›.



‹(a,b)A(c)(d,e)›

(a,b)A(c) est entouré de crochets omis. En effet l'arbre



possède 3 sous-arbres: (a,b)A(c), d, et e.

figure 6.1

6.3.2 Aspects dynamiques

6.3.2.1 Recherche de l'‹oprt-id› de plus forte priorité

Lorsque dans une expression entre crochets explicites ou implicites, plusieurs ‹op-call› sont placés entre crochets omis, c'est l'‹op-call› dont l'‹oprt-id› a la plus forte priorité qui

est évalué.
Ensuite tous les autres ‹op-call› entre crochets omis sont évalués dans l'ordre des priorités croissantes et non pas séquentiellement de gauche a droite (qui est l'ordre dans lequel ils seraient évalués s'ils étaient entourés par des crochets explicites).

Les différents ‹op-call› entre crochets omis d'une expression entre crochets explicites ou implicites appartiennent à la forêt d'un même arbre. Chaque arbre de cette forêt est connu dans la mesure où les atomes qui constituent chaque arbre ont été extraits. (Ce ne serait pas nécessaire s'ils étaient évalués séquentiellement de gauche a droite. Il suffirait alors de les évaluer au fur et à mesure qu'ils sont extraits.)

> Exemple: ‹(...)A(..)(...)B(...)› Supposons A de plus forte priorité que B alors ‹(...)A(...)› est évalué en premier lieu. Si (...)A(...) généré (...)C(...) alors (...)C(...) se substitue à (...)A(...) pour donner ‹(...)C(...)(...)B(...)›. C'est le plus prioritaire de C ou de B qui sera ensuite évalué.

6.3.2.2 Omission de parenthèses

Remarques: (règles 5)
1.Si le nombre d'arguments présents dans l'‹op-call› d'un opérateur est inférieur au nombre d'arguments utilisés par l'opérateur, les arguments manquants sont considérés comme vide.
Exemple:

‹A=(...‹ARG(4)›...)›

‹A(a,b,c)›

‹ARG(4)› génère l'argument vide.

2.Deux ‹arg-sep› consécutifs, une parenthèse ouvrante et un

‹arg-sep›, un ‹arg-sep› et une parenthèse fermante délimitent un ‹arg› vide.

Exemple:

‹A(,,)›

L'‹op-call› possède 3 ‹arg› qui sont tous les trois vides.

3.Si un ‹oprt-id› est suivi (précédé) immédiatement d'un ‹arg-sep› et s'il possède au moins un argument droit (gauche) alors cet argument est vide.(PEGASE p.9)

Exemple:

‹..., B ,...› avec ‹B=(1,1,...)› possède deux arguments vides.

Pour construire l'arbre d'un ‹op-call› dont les ‹arg-seq› sont délimités par des parenthèses omises, il est nécessaire de connaître le nombre maximum d'opérandes gauches et droits.
En appliquant la règle 5, les ‹arg› de l'‹oprt-id› de plus forte priorité peuvent être déterminés.(figure 6.2)

‹A=(0,1,1,...)›

‹a,b A c,d›

L'arbre de cet ‹op-call› est:

L'‹op-call› ‹A(c)› sera évalué en premier lieu.

figure 6.2

On est sûr que tous les ‹arg› de l'‹oprt-id› de plus forte priorité ont été extraits puisque tous les atomes compris entre crochets explicites ou implicites doivent être extraits avant de

choisir l'‹oprt-id› de plus forte priorité. En effet si les ‹arg-seq› de l'‹oprt-id› selectionné sont délimités par des parenthèses, ces parenthèses sont incluses dans les crochets explicites ou implicites. Si les parenthèses de l'‹arg-seq› sont omises, les ‹arg› de l'‹arg-seq› gauche (droite) sont comprises entre l'‹oprt-id› sélectionné et l' ‹oprt-id› suivant, ou à défaut le crochet explicite ou implicite ouvrant (fermant) suivant.

6.3.2.3 Remarques

1.L'évaluation d'un ‹op-call› sous-arbre d'un autre arbre peut modifier cet arbre (figure 6.3)
Ce mécanisme existe également avec des parenthèses explicites.
Exemple :soit ‹A=(0,1,1,‹VAL("c")›)›
l'évaluation de ‹(a,b)A(c)(d)› donnera
‹(a,b)C(d)›

Le deuxième ‹image-programme› ‹a,b C d› possède un crochet omis fermant que ne possède pas l'‹image-programme› ‹a,b A c d› entre l'‹arg› d et le crochet fermant explicite. Ce crochet fermant omis et le crochet ouvrant correspondant délimite un sous-arbre constitué par C et d; c'est pourquoi d passe à un niveau supérieur du graphe.

Les arbres représentent la partie de l'‹image-programme› dont les atomes ont déjà été extraits mais dont les ‹op-call› n'ont pas encore été évalués. Lorsque l'évaluation d'un ‹op-call› fait apparaitre un crochet omis à un endroit du texte déjà atomisé, la répartition des ‹ARG› entre les niveaux se modifie.

2.Nous avons toujours considéré jusqu'ici qu'un ‹op-denot› du langage IMAGE pouvait n'avoir qu'une ‹arg-seq› parenthétisée gauche ou droite ou même pas de ‹arg-seq› parenthétisée du tout

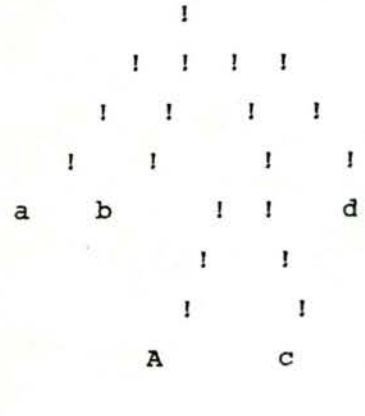
alors

soit ‹A=(0,1,1,‹VAL("C")›)
‹C=(0,1,1,...)›

L'évaluation de ‹a,b A c d›

donnera ‹a,b C d›

Nous passons de l'arbre



a l'arbre

figure 6.3

que la syntaxe de IMAGE impose une (‹arg-seq›) gauche et droite même si ‹arg-seq› est vide.

‹op-denot›::=(‹arg-seq›)‹oprt-id›(‹arg-seq›)
‹arg-seq›::=‹arg›!‹arg›‹arg-sep›‹arg-seq›!‹empty›

L'hypothese que les ‹arg-seq› parenthétisées ne sont pas obligatoires est justifiée par le langage MAGE. Il suffit de déclarer le nombre maximum d'‹arg› gauches et droits égal à zéro et d'omettre les paires de parenthèses.

6.4 Troisième partie

Nous repartons ici de l'analyse faite pour le langage IMAGE et nous allons y ajouter les éléments nécessaires pour que l'interpréteur MAGE puisse interpréter un texte en langage MAGE.

Chaque fois que nous parlerons de crochets dans la pile des crochets nous voudrons dire le pointeur dû à un crochet.
Le pointeur dû à un crochet ouvrant (explicite ou implicite) est le pointeur vers le premier élément ajouté à la liste après un crochet ouvrant explicite ou implicite.

6.4.1 détection des crochets implicites

Nous savons que dans toute structure associée à un ‹image-programme› existe une pile des crochets. Nous savons aussi que outre les crochets ouvrants cette pile mémorise les parenthèses ouvrantes.
Il est maintenant nécessaire que cette pile mémorise les crochets ouvrants implicites et explicites ainsi que les parenthèses ouvrantes: l'interpréteur pourra toujours y distinguer les crochets explicites, les crochets implictes et les parenthèses.

Chaque fois qu'il lira un ‹crlf›, l'interpréteur devra détecter s'il s'agit d'un crochet fermant implicte.
La condition nécessaire et suffisante pour qu'un ‹crlf› soit un crochet fermant implicite est qu'au sommet de la pile des crochets ouvrants soit mémorisé un crochet ouvrant implicite. (règle 2) En effet si le ‹crlf› lu n'était pas un crochet ouvrant implicite, ce ‹crlf› serait entre deux parenthèses (crochets explicites) et il y aurait au sommet de la pile une parenthèse ouvrante (un crochet ouvrant explicite) ou bien la pile serait

vide car aucun crochet implicite n' était ouvert. Ceci montre bien pourquoi il faut mémoriser dans la pile les crochets et parenthèses ouvrantes et pourquoi l'interpréteur doit pouvoir y distinguer les crochets implicites des explicites.

Un ‹crlf› sera un crochet ouvrant si au sommet de la pile des crochets il n'y a ni parenthèse ni crochet ouvrant et si de plus le ‹crlf› n'est pas suivi par un crochet ouvrant explicite ou une suite de labels elle-même suivie d'un crochet ouvrant explicite. (règle 1) Ceci signifie qu'un atome supplémentaire devra être extrait après un ‹crlf› s'il n'existe pas de crochet ou de parenthèse ouvrantes au sommet de la pile. Si cet atome n'est ni un crochet explicite ni un label alors, le ‹crlf› est un crochet ouvrant implicite.

6.4.2 Evaluation des ‹op-call›

6.4.2.1 Début de l'évaluation

L'interpréteur extrait les atomes de l'‹image-programme› en cours jusqu'au moment où il rencontre un crochet fermant explicite ou implicite. C'est seulement alors que les différents ‹op-call› entre crochets omis compris entre le dernier crochet ouvrant non encore fermé (implicite ou explicite) extrait et la position courante pourront être évalués.

6.4.2.2 Notion de sous-liste

Le dernier atome extrait étant un crochet fermant(explicite ou implicite), nous appellerons sous-liste la suite des éléments de

la liste courante (celle à laquelle les atomes actuellement extraits sont ajoutés) comprise entre le crochet ouvrant (explicite ou implicite) au sommet de la pile et l'élément dont le pointeur est précédent(position courante). Tous les ‹op-call› entre crochets omis contenus dans cette sous-liste devront être évalués avant de continuer l'extraction des atomes de l'‹image-programme› en cours.

Exemple:
‹crlf›...‹crlf›
1 2
pile des crochets: 1
Lorsque la position courante sera 2 l'évaluation des ‹op-call› de la sous-liste comprise entre les positions 1 et 2 pourra commencer.

6.4.2.3 Détermination des limites extrêmes des ‹op-call› d'une sous-liste

Il s'agit d'évaluer les uns à la suite des autres tous les ‹op-call› contenus dans une sous-liste. Il faut d'abord déterminer l'‹oprt-id› non compris entre parenthèses de plus forte priorité et le plus à gauche. Ensuite il faut connaitre le pointeur du premier et du dernier élément de l'‹op-call› à évaluer.

Nous appellerons limite gauche le pointeur du premier élément et limite droite le pointeur du dernier élément de l'‹op-call› à évaluer.
Pour déterminer l'‹oprt-id›, l'interpréteur balaye la sous-liste de gauche à droite. Pour chaque élément dont la classe syntaxique est $ident ou dont l'atome est un ‹op-char›, il consulte la table des symboles. Si le type de l'objet associé au symbole est $eprg, $eprgv, $iprg ou $iprgv et que la priorité associée à ce symbole est plus forte que celle des éléments rencontrés jusque

là dans la sous-liste, il retient le pointeur de l'atome associé à cet ‹oprt-id›. Lorsque toute la sous-liste est parcourue, le dernier ‹oprt-id› retenu est le plus à gauche parmi les opérateurs de priorité maximum de la sous-liste.

6.4.2.4 Fin de l'évaluation des ‹op-call› d'une sous-liste

L'évaluation des ‹op-call› d'une sous-liste est terminée lorsque après avoir balayé la sous-liste, l'interpréteur ne trouve plus d'‹oprt-id› exécutable. Alors le crochet ouvrant (explicite ou implicite) est éliminé du sommet de la pile des crochets et l'extraction reprend à partir de la position courante.

6.4.2.5 Génération des textes

6.4.2.5.1 ‹sys-ident›, opérateur à et opérateur ARG

Pour les ‹sys-ident› et les suites d'éléments générés par les opérateurs à et ARG rien n'est changé. (5.3.5.2, 5.3.5.3, 5.3.5.5)
Le ‹sys-ident› pointe à gauche vers l'atome dont le pointeur est précédent(limite gauche) et à droite vers l'atome dont le pointeur est suivant (limite droite).
Le premier élément d'une suite d'éléments pointe vers l'élément dont le pointeur est précédent(limite gauche). Le dernier élément d'une suite d'éléments pointe vers l'élément dont le pointeur est suivant(limite droite).

Exemples:

A.Génération d'un ‹sys-ident›

.L'‹op-call› qui génère le ‹sys-ident› sysid est entre les crochets explicites.

‹A=(2,1,2,...)›

‹B=(1,1,1,...)›

‹cr›a A‹b B c›,d‹cr›

La structure avant l'évaluation de ‹b B c› sera:

```
liste:  a__A__b__B__c__,d‹cr›
        1     2        3
position courante:  3
pile des crochets:  2 ‹
                    1 ‹cr›
```

La structure après évaluation de ‹b B c› est:

```
liste:  a__A__sysid__,d‹cr›
        1     2      3
position courante:  3
pile des crochets:  1 ‹cr›
```

.L'‹op-call› qui génère le ‹sys-ident› sysid est entre crochets implicites

‹A=(1,1,3,...)›

‹C(1,1,1,...)›

‹F=(2,2,2,...)›

‹cr›A a,b C d,e F‹cr›...

L'évaluation de C génère le ‹sys-ident› sysid.

La structure avant l'évaluation de ‹b C d› est:

```
liste:  A__a__,__b__C__d__,__e__F__...
        1        3     4           2
position courante:  2
pile des crochets:  1 ‹cr›
limite gauche:  3
limite droite:  4
```

La structure après l'évaluation de ‹b C d›:

```
liste:  A__a__,__sysid__,__e__F__...
        1        3               2
position courante:  2
pile des crochets:  1 <cr>

B.Opérateur ARG
<A=((a ARG 1 b<cr>...)>
Soit A l'<image-programme> en cours suite à l'<op-call>
<cr>A((a)B(c),d)<cr>
La structure avant l'évaluation de ARG 1 est:
liste: a__ARG__1__b__...
       1  2    3     4
position courante:  4
pile des crochets:  1 <cr>
limite gauche:  2
limite droite:  3

La structure après l'évaluation de ARG 1 sera:
liste:  a__(__a__)__B__(__c__)__b__...
        1                          4
position courante:  4
pile des crochets:  1 <cr>

C.Opérateur à

Soit B l'<oprt-id> d'un <image-programme>.
Puisqu'il s'agit d'un <image-programme> l'évaluation
de l'<op-call> auquel cet <oprt-id> appartient suspend
la struture associée à l'<image-programme>
contenant cet <op-call>.
<B=(0,0,1...)>
<cr> a B c <cr>
La structure lorsqu'elle est suspendue:
liste:  a__B__c__...
        1  2     3
position courante:  3
pile des crochets:  1 <cr>
```

limite gauche: 2

limite droite: 2

Apres l'interprétation de l'‹image-programme› de l'opérateur B, la suite des éléments générés par l'opérateur à se substituent à l'‹op-call› et la strucutre précédente est réactivée .

Structure lorsqu'elle est réactivée:

liste: a__suite des atomes générés par à__c__...
1 3

position courante: 3

pile des crochets: 1 ‹cr›

6.4.2.5.2 Opérateur VAL

Seul l'opérateur VAL génère un texte qui n'est pas une suite d'éléments faisant partie d'une liste. Il génère un ‹string› dont il faut extraire les atomes qui le constituent.(5.3.3.3) Dans le langage MAGE, l'‹op-call› auquel le ‹string› doit se substituer peut être n'importe où dans une sous-liste.

Exemple:

‹string="a A b"›

‹cr›a,b VAL string c,d‹cr›...

La strucutre avant l'évaluation de VAL string est:

liste: a__,__b__VAL__string__c__,__d__...
1 3 4 2

position courante: 2

pile des crochets: 1 ‹cr›

limite gauche: 3

limite droite: 4

La structure après l'évaluation de VAL string sera:

liste: a__,__b__a A b__c__,__d__...

1 2

position courante: 2

pile des crochets: 1 ‹cr›

A ce moment a A b constitue un seul atome.

Avant de chercher l'‹oprt-id› de plus forte priorité suivant dans la sous-liste, il est nécessaire d'extraire les atomes du texte substituant. Il est en effet possible que l'‹oprt-id› exécutable suivant soit contenu dans ce ‹string›.(Le ‹string› peut être un ‹op-call› entre crochets omis comme le montre l'exemple précédent.) Le ‹string› généré peut également faire apparaitre une nouvelle sous-liste à l'intérieur de la précédente.

Exemple:

‹string="A›e"›

‹‹a,b,VAL string,c,d›...

ce qui après évaluation de VAL string donnera

‹‹a,b,A›e,c,d›...

La sous-liste ‹a,b,A› doit être complètement exploitée avant de terminer celle qui la contient. L'interpréteur devra extraire des atomes avec pour nouvelle position courante le pointeur vers ‹string› généré. L'ancienne position courante reste en sommeil jusqu'au moment où tous les atomes du ‹string› généré ont été extraits.

Nous proposons de mémoriser les positions courantes dans une pile appelée pile des positions courantes.

L'extraction des atomes se fera toujours du texte de l'atome dont le pointeur est au sommet de cette pile. Nous dirons que le pointeur de cet élément est sommet(pile des positions courantes).

Quand VAL génère un atome, le pointeur vers celui-ci est mémorisé au sommet de la pile des positions courantes.

Quand le texte de l'élément dont le pointeur est sommet(pile des positions courantes) est vide, cet élément est éliminé de la liste et son pointeur est retiré du sommet de la pile. Ainsi l'extraction des atomes continue à partir de l'atome dont le pointeur est sommet(pile des positions courantes). Evidemment

chaque structure possède sa propre pile des positions courantes.

Ceci explique le point laissé en suspens lorsque nous avons parlé de l' opérateur VAL dans le langage IMAGE. (5.3.5.4) La pile des positions courantes aurait été une solution possible pour l'opérateur VAL dans le langage IMAGE. Cependant cette solution était trop compliquée. Il suffisait de faire un seul atome du ‹string› généré et du texte non encore interprété en les concaténant. La position courante aurait été le pointeur vers ce nouvel élément. Par contre pour l'opérateur VAL dans le langage MAGE cette solution n'est pas acceptable puisque le ‹string› peut être généré n'importe où dans la sous-liste. C'est pourquoi nous ne l'avons pas proposé lorsque nous avons discuté du langage IMAGE.

Reprenons l'exemple:

```
        ‹string = "A›e">
        ‹‹a,b,VAL string,c,d›f,g F
        La structure avant l'évaluation de VAL string est:
        liste:  a__,__b__,__VAL__string__,__c__,__d__f,g F
                1           3    4                   2
        pile des positions courantes:  2
        pile des crochets:  1
                            1
        limite gauche:  3
        limite droite:  4
        La structure après l'évaluation de VAL string et
        avant l'extraction des atomes du ‹string› généré sera:
        liste:  a__,__b__,__A›e__,__c__d__f,g F
                1           5             2
        pile des positions courantes:  5
                            2
        pile des crochets:  1
                            1
        Structure lorsque la nouvelle sous-liste apparait:
```

```
liste:  a__,__b__,__A__e__,__c__d__f,g F
        7           6  5           2
pile des positions courantes:  5
                               2
pile des crochets:  1
                    1
Si A est défini comme ‹A=(3,0,..)›
limite gauche:  7
limite droite:  6
Structure lorsque la nouvelle sous-liste disparait:
liste:  ...__e__,__c__d__f,g F
        1    5           2
pile des positions courantes:  5
                               2
pile des crochets:  1
Structure quand tous les atomes du ‹string›  généré
ont été extraits:
liste:  ...__e__,__c__d__f,g F
        1                2
pile des positions courantes:  2
pile des crochets:  1
```

7.DESCRIPTION DE L'INTERPRETEUR

7.1 Introduction

Dans ce chapitre, nous allons donner une approche plus algorithmique de l'interpréteur MAGE.
Les algorithmes s'appuyeront sur les concepts et les objets décrits dans les chapitres précédents. Si nécessaire, nous en rappellerons brièvement l'un ou l'autre.

Le premier algorithme qui sera décrit proposera une démarche générale pour interpréter un programme en langage MAGE.
Le deuxieme montrera comment retrouver l'‹oprt-id› de plus forte priorité le plus à gauche dans une sous-liste.
Le troisieme algorithme déterminera la limite gauche de l'‹op-call› à évaluer. (L'algorithme qui déterminerait la limite droite serait son symetrique.)

7.2 Algorithme d'interprétation.

7.2.1 Quelques précisions

L'extraction des atomes se fait à partir de l'atome de l'élément dont le pointeur est sommet(pile des positions courantes). Pour que l'atome extrait soit vide il est nécessaire non seulement que atome(sommet(pile des positions courantes)) soit vide mais aussi que sommet(pile des positions courantes) soit le seul pointeur dans la pile des positions courantes. Ces deux conditions signifient que l'interprétation de l'‹image-programme› en cours

est terminée.

La distinction entre crochet implicite et crochet explicite n'influence en rien la façon de gérer la liste des atomes; c'est pourquoi l'algorithme ne fait pas la distinction.

Le fait qu'un ‹oprt-id› n'a pu être déterminé dans une sous-liste peut être dû à ce que la sous-liste est vide.
Une sous-liste est vide si
suivant(précédent(sommet(pile des crochets)))
=sommet(pile des positions courantes).
Ceci suppose évidemment que l'information précédent(sommet(pile des crochets)) n'a pas été perdue.
Un ‹oprt-id› exécutable peut aussi ne pas exister alors que la sous-liste n'est pas vide. Nous pourrions considérer cela comme une erreur. Cependant dans un tel cas l'interprétation peut continuer en passant à la sous-liste suivante. C'est ce que fait l'algorithne.

La négation de la condition "un ‹oprt-id› exécutable existe" couvre deux cas:

- -la sous-liste est vide
- -la sous-liste n'est pas vide mais ne contient pas d'‹oprt-id› qui ne soit pas entre parenthèses.

Implicitement avant chaque test de cette condition, l'‹oprt-id› de plus forte priorité le plus à gauche sera recherché dans la sous-liste.

7.2.2 Rappel

-Une sous-liste est une partie de la liste. Le pointeur du premier élément de la sous-liste en cours est sommet(pile des crochets) et l'adresse de son dernier élément est

précédent(sommet(pile des positions courantes)). (6.4.2.2)
-Les limites gauches et droites sont les adresses dans une liste du premier et du dernier élément de l'‹op-call› à évaluer. (6.4.2.3)

7.2.3 Initialisation d'une structure

Au moment d'évaluer un opérateur IMAGE, il faut
-suspendre la structure existante
-créer et initialiser une nouvelle structure. Cette initialistion consiste à:

-initialiser une pile des positions courantes avec le pointeur vers l'élément dont l'atome mémorise le texte du nouvel ‹image-programme›.
-mémoriser au niveau de la nouvelle structure les adresses
-limite gauche et
-limite droite

Ainsi chaque structure connaitra le pointeur du premier et du dernier atome de l'‹op-call› ayant appelé l'‹image-programme› qui lui correspond. Ceci permettra le moment venu d'insérer la suite des atomes générés par l'opérateur à.
De plus l'opérateur ARG saura entre quelles limites aller chercher l'‹arg› désigné par son opérande.

7.2.4 Limites gauches et droites

Nous venons de dire plus haut que lors de l'appel d'un opérateur IMAGE les limites droites et gauches étaient mémorisées dans la

structure de l'‹image-programme› associée à cet opérateur IMAGE.
Lors de l'appel d'un opérateur de base, l'interpréteur leur communique les limites gauches et droites.
L'opérateur de base peut ainsi retrouver ses arguments et lorsqu'il ne génère rien éliminer son ‹op-call› de la liste:

suivant(précédent(limite gauche))=suivant(limite droite)

précédent(suivant(limite droite))=précédent(limite gauche).

Grâce aux limites gauche et droite, les opérateurs ARG et VAL peuvent insérer correctement la suite des atomes qu'ils génèrent et qui se substitue à leur ‹op-call›.

7.2.5 Pile des positions courantes

Lorsque l'opérateur VAL est appelé, il ajoute au sommet de la pile des positions courantes le pointeur vers l'atome qu'il substitue dans la liste à son ‹op-call›.

7.2.6 Pointeur ^

Dans l'algorithme, ^ désigne le pointeur vers le prochain élément qui sera inséré dans une liste.

7.2.7 Algorithme (figure 7.1)

7.3 Recherche de l'‹oprt-id› exécutable de plus forte priorité

```
-extraire l'atome suivant
-si l'atome extrait est non vide
  -si l'atome extrait est un crochet fermant
    -si pile des crochets est vide ou si
    sommet(pile des crochets) est une parenthèse
      -erreur
    -tant qu'un ‹oprt-id› exécutable existe
      -déterminer la limite gauche
      -déterminer la limite droite
      -si le type de l'‹oprt-id› est $iprg ou $iprgv
        -stacker la structure en cours
        -créer et initialiser une nouvelle structure
      -si le type de l'‹oprt-id› est $eprg ou $eprgv
        -appeler l'opérateur en lui communiquant
        les limites gauches et droites
    -éliminer le pointeur au sommet de la pile des crochets
 sinon -si l'atome extrait est un crochet ouvrant
         -sommet(pile des crochets):=^
       sinon -insérer l'atome dans la liste devant l'atome
             dont le pointeur=sommet(pile des positions courantes)
sinon -si la pile des structures est non vide
        -substituer la suite des atomes générés par
        l'opérateur à à l'‹op-call› de l'‹image-programme›
        associé à la structure en cours
        -la structure du sommet de la pile des structures
        devient la structure en cours
      sinon stop
```

figure 7.1

Il s'agit ici de trouver dans la sous-liste courante l'‹oprt-id› ‹ident› ou ‹op-char› le plus à gauche de plus forte priorité et qui ne soit pas entre deux parenthèses. Rappelons que le pointeur vers le premier élément de la sous-liste courante est sommet(pile des crochets) tandis que le pointeur vers son dernier élément est précédent(sommet(pile des crochets)). (6.4.2.2)

L'algorithme (figure 7.2) suppose que la sous-liste est non vide c.a.d. que suivant(précédent(sommet(pile des crochets)))
£sommet(pile des positions courantes)
ou que
précédent(sommet(pile des positions courantes))
£sommet(pile des crochets).

L'algorithme consiste à balayer la sous-liste de gauche à droite et à retenir l'‹oprt-id› de plus forte priorité le plus à gauche. Lorsque toute la sous-liste a été parcourue, c'est l'‹op-call› du dernier ‹oprt-id› retenu qui est à évaluer.

Nous prendrons les conventions suivantes:
A: pointeur vers l'élément courant
C: priorité de l'‹oprt-id› retenu
D: pointeur vers l'‹oprt-id› retenu
type(symbole) donne le type associé à un objet dans la table des symboles
priorité(symbole) donne la priorité associée à un ‹oprt-id› dans la table des symboles.

Dans l'algorithme, chercher) permet de sauter la partie de la sous-liste incluse dans des parenthèses:

```
chercher ):
-y:=1
-jusqu'à ce que y=0
        -A:=suivant(A)
        -si atome(A)=)
                -y:=y-1
        sinon   -si atome (A)=(
                -y:=y+1
```

(‹op-char›) est l'ensemble des ‹op-char› de la syntaxe.

```
-C:=0
-A:=sommet(pile des crochets)
-B:=sommet(pile des positions courantes)
-jusqu'à ce que A=B
 -si atome(A)=(
  -chercher )
 sinon -si classe syntaxique(A)=$ident
        -si type(atome(A))=$eprg ou $eprgv ou $iprg
        ou $iprgv
         -si priorité(atome(a))>C
          -D:=A
          -C:=priorité(atome(A))
        sinon -si atome(A) appartient à (<op-char>)
          -D:=A
          -C:=priorité(atome(A))
 -A=suivant(A)
```

figure 7.2

7.4 Recherche de la limite gauche

Le pointeur de l'<oprt-id> à exécuter déterminé, il reste à connaitre les limites gauches et droites de l'<op-call> de cet <oprt-id>. (L'algorithme de recherche de la limite droite est son symetrique.)

L'algorithme parcourt la sous-liste de droite à gauche à partir de l'<oprt-id> jusqu'au moment ou l'<arg-seq> parenthetisé gauche de cet <oprt-id> est determinée.

7.4.1 <oprt-id> <arg-sep> parenthèses et arguments

Les atomes d'une liste peuvent avoir les classes syntaxiques

suivantes:

$spident

$ident

$strden

$numden.

Ils peuvent également être des ‹op-char› des ‹arg-sep›, des ‹sys-ident› ou des parenthèses.

Dans l'algorithme suivant nous distinguerons les atomes d'une liste qui sont des ‹oprt-id› ou des ‹arg-sep› ou des parenthèses des autres atomes que nous appellerons argument.

7.4.2 Rappel

L'‹oprt-id› retenu ne peut être entre des parenthèses. S'il l'était, il ne serait pas un ‹oprt-id› exécutable.

7.4.3 Conventions

Soient:

X: nombre d'arguments déjà parcourus

Z: pointeur vers l'élément courant

A: pointeur du premier atome de la sous-liste

Ad: pointeur vers l'‹oprt-id› de l'‹op-call› dont on recherche la limite gauche

Ga: nombre maximum d'arguments gauches

La condition:

(classe syntaxique(Z)=$ident et type(atome(Z))=$iprg ou $iprgv ou $eprg ou $eprgv) ou atome appartient à (‹op-char›)

sera écrite élément(Z) est un ‹oprt-id›.

La condition:

(classe syntaxique(Z)=$spident ou $strden ou $numden) ou

(classe syntaxique(Z)=$ident et type(atome(Z))£$eprg ou $eprgv ou$iprg ou $iprgv) ou atome(Z) est un ‹sys-ident› sera écrite élément(Z) est un argument.

7.4.4 Fin de l'algorithme

Pour terminer sa recherche, l'algorithme doit avoir parcouru un nombre d'arguments égal à Ga (c'est le test X=Ga) ou bien sortir de la sous-liste (test Z=sommet(pile des crochets)) ou encore trouver un ‹oprt-id›. Tant qu'une de ces conditions ne sera pas satisfaite, il parcourra le schéma suivant:

Devant un ‹oprt-id› l'algorithme peut trouver

A. un ‹oprt-id› et la recherche est terminée

B. un argument. Devant argument‹oprt-id› peuvent exister:

- B1. un ‹oprt-id› et la recherche est terminée
- B2. un argument. Il y a alors erreur.
- B3. un). Il y a alors erreur.
- B4. un ‹arg-sep›. Il faut continuer en D.

C. un). Il faut alors chercher la parenthèse fermante. La parenthèse trouvée, la recherche est terminée.

D. un ‹arg-sep›. Devant ‹arg-sep›‹oprt-id› peuvent exister:

- D1. un ‹oprt-id› et la recherche est terminée.
- D2. un argument et il faut continuer en B.
- D3. un). Il faut chercher la (correspondante puis aller en B.
- D4. un ‹arg-sep›. Il faut alors continuer en D.

Chercher la parenthèse correspondante peut s'écrire sous forme d'un algorithme:

chercher (:

    y:=1

```
        jusqu'à ce que y=0
                Z:=précédent(Z)
                si atome(Z)=(
                        y:=y-1
                sinon   si atome(Z)=)
                                y:=y+1
```

7.4.6 Algorithme

```
Z:=précédent(ad)
si Z=sommet(pile des crochets)
        aller à la sortie
si élément(Z) est un ‹oprt-id›
        limite gauche:=suivant(Z)
        aller à la sortie
si élément(Z) est un argument
        X:=X+1
        Si X=Ga
                limite gauche:=Z
                aller à la sortie
        sinon   Z:=précédent(Z)
                aller en 1
si atome(Z) est une )
        chercher (
        limite gauche:=Z
        aller à la sortie
si atome(Z) est un ‹arg-sep›
        X:=X+1
        si X=Ga limite gauche:=suivant(Z)
                aller à la sortie
        sinon   Z:=précédent(Z)
                aller en 3

1: si Z=sommet(pile des crochets)       aller à la sortie
si élément(Z) est un ‹oprt-id›
        limite gauche:=suivant(Z)
        aller à la sortie
```

```
si élément(Z) est un argument ou atome(Z)=)
        erreur
si atome(Z) est un ‹arg-sep›
        Z:=précédent(Z)
        aller en 3

3: si Z=sommet(pile des crochets)       aller à la sortie
si élément(Z)est un ‹oprt-id›
        limite gauche:=suivant(Z)
        aller à la sortie
si élément(Z) est un argument
        X=Z+1
        si X=Ga
                limite gauche:=Z
                aller à la sortie
        sinon   Z:=précédent(Z)
                aller en 1
si atome(Z)=)
        chercher (
        X:=X+1
        si X=Ga
                limite gauche: =Z
                aller à la sortie
        sinon   Z:=précédent(Z)
                aller en 1
si atome(Z) est un ‹arg-sep›
        X:=X+1
        si X=Ga
                limite gauche:=suivant(Z)
                aller à la sortie
        sinon Z:=précédent(Z)
                aller en 3
sortie:
```

8. CONCLUSIONS

Nous avons simplifié l'analyse de l'interpréteur MAGE en passant par une étape intermédiaire: l'étude de l'apport spécifique du langage IMAGE.

L' analyse nous a permis de décrire les objets manipulés par l'interpréteur (les structures, les listes, les sous-listes, la pile des crochets, la pile des positions courantes...).
Nous avons étudié plus particulièrement les opérateurs qui genèrent du texte (ARG, VAL, à).
Nous avons enfin décrit un algorithme général pour interpréter un programme en langage MAGE et nous avons décrit des algorithmes de recherche de l'‹oprt-id› de plus forte priorité et de ses ‹arg›.

Il reste à poursuivre l'analyse de façon de plus en plus détaillée:

- -il faut décrire comment chaque opérateur manipule les objets de l'interpréteur;
- -l'implémentation des objets de l'interpréteur reste à préciser;
- -le traitement des erreurs est un point important à ne pas négliger.

La programmation serait faite parallèlement à cette analyse en s'appuyant sur les algorithmes que nous avons proposés.

Pour terminer viendrait le temps de l'implementation et des tests. Disposant après cela de l'interpréteur MAGE, nous pourrions mieux juger le langage MAGE.